

TDDI16 – Föreläsning 3

Symboltabeller och träd

Filip Strömbäck

Planering

Vecka	Fö	Lab
36	Komplexitet, Linjära strukturer	----
37	Träd, AVL-träd	1---
38	Hashning	12--
39	Grafer och kortaste vägen	12--
40	Fler grafalgoritmer	-23-
41	Sortering	--3-
42	Mer sortering, beräkningsbarhet	--34
43	Tentaförberedelse	---4

- 1 ADT dictionary (symboltabell)
- 2 Binära sökträd
- 3 AVL-träd, ett balanserat sökträd
- 4 Sammanfattning

Problem

Du håller på att skriva en rapport, och för att se till att göra ett gott intryck vill du se till att du inte har några stavfel i rapporten. Du har hittat en bra ordlista på internet (och i `/usr/share/dict/words`), och tänker att det inte kan vara särskilt svårt att skriva ett program som kontrollerar alla ord i rapporten efter stavfel.

Ordlistan innehåller n ord, rapporten innehåller m ord.

Hur kan vi göra detta på ett bra sätt?

Hur svårt kan det vara?

```
int main() {  
    vector<string> d = read_dict();  
  
    int count{};  
    string word;  
    while (cin >> word) {  
        if (find(d.begin(), d.end(), word) == d.end())  
            cout << "Wrong: " << word << endl;  
        count++;  
    }  
    cout << "Checked " << count << " words" << endl;  
    return 0;  
}
```

Testkörning

```
$ time spellcheck < thesis.txt
99171 words in the dictionary
...
Checked 38040 words

real 0m36.728s
user 0m36.663s
sys 0m0.047s
```

Inte jättebra...

Analys – find

```
template <typename Iter, typename Elem>
Iter find(Iter begin, Iter end, Elem elem) {

    for (Iter i = begin; i != end; ++i) {
        if (*i == elem) {
            return i;
        }
    }
    return end;
}
```

Analys – find

```
template <typename Iter, typename Elem>
Iter find(Iter begin, Iter end, Elem elem) {

    for (Iter i = begin; i != end; ++i) {
        if (*i == elem) {
            return i;
        }
    }
    return end;
}
```

$\left. \begin{array}{l} \text{for loop body} \\ \text{if statement} \\ \text{return statement} \end{array} \right) \mathcal{O}(1)$ $\left. \vphantom{\begin{array}{l} \text{for loop body} \\ \text{if statement} \\ \text{return statement} \end{array}} \right) n \text{ gånger}$

Analys – find

```
template <typename Iter, typename Elem>
Iter find(Iter begin, Iter end, Elem elem) {

    for (Iter i = begin; i != end; ++i) {
        if (*i == elem) {
            return i;
        }
    }
    return end;
}
```

$\left. \begin{array}{l} \text{for (Iter i = begin; i != end; ++i) \{ } \\ \quad \text{if (*i == elem) \{ } \\ \quad \quad \text{return i; } \\ \quad \} \\ \} \end{array} \right) \mathcal{O}(1) \quad n \text{ gånger}$

Totalt: $\mathcal{O}(n \cdot 1) = \mathcal{O}(n)$ Vad är *bästa* och *värsta* fall?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall $\Theta(1)$

`find`, värsta fall $\Theta(n)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall $\Theta(1)$

`find`, värsta fall $\Theta(n)$

`find`, medelfall $\Theta(n)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall $\Theta(1)$ $\mathcal{O}(1)$ $\Omega(1)$

`find`, värsta fall $\Theta(n)$ $\mathcal{O}(n)$ $\Omega(n)$

`find`, medelfall $\Theta(n)$ $\mathcal{O}(n)$ $\Omega(n)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Bästa och värsta fallet

Vad kan vi säga om bästa och värsta fallet på `find`?

`find`, bästa fall $\Theta(1)$ $\mathcal{O}(1)$ $\Omega(1)$

`find`, värsta fall $\Theta(n)$ $\mathcal{O}(n)$ $\Omega(n)$

`find`, medelfall $\Theta(n)$ $\mathcal{O}(n)$ $\Omega(n)$

`find` - $\mathcal{O}(n)$ $\Omega(1)$

Vad är **relevant** att säga? Vad är **tillräckligt** att säga?

Analys

```
while (cin >> word) {  
    if (find(...) == d.end())  $\mathcal{O}(n)$   
        cout << ...;  $\mathcal{O}(1)$   
}
```

$\left. \vphantom{\begin{array}{l} \mathcal{O}(n) \\ \mathcal{O}(1) \end{array}} \right) m \text{ gånger}$

Analys

```
while (cin >> word) {  
    if (find(...) == d.end())  $\mathcal{O}(n)$   
        cout << ...;  $\mathcal{O}(1)$   
}
```

$\left. \vphantom{\begin{array}{l} \mathcal{O}(n) \\ \mathcal{O}(1) \end{array}} \right) m \text{ gånger}$

Totalt: $\mathcal{O}(m \cdot (1 + n)) = \mathcal{O}(m + mn) = \mathcal{O}(mn)$

Analys

```
while (cin >> word) {  
    if (find(...) == d.end())  $\mathcal{O}(n)$   
        cout << ...;  $\mathcal{O}(1)$   
}
```

) m gånger

Totalt: $\mathcal{O}(m \cdot (1 + n)) = \mathcal{O}(m + mn) = \mathcal{O}(mn)$

Om $m \approx n \implies \mathcal{O}(n^2)$: bra eller dåligt?

Analys – vad behövs?

Analys – vad behövs?

	vector
Insättning	
Medlemstest	
Ladda ordlista	
Stavningskontroll	

Analys – vad behövs?

	vector	Önsketänkande
Insättning	$\Theta(1)$ ¹	
Medlemstest	$\mathcal{O}(n)$	
Ladda ordlista	$\Theta(n)$	
Stavningskontroll	$\mathcal{O}(mn)$	
$\approx$	$\mathcal{O}(n^2)$	

¹Medelfall

Analys – vad behövs?

	vector	Önsketänkande
Insättning	$\Theta(1)^1$	$\Theta(1)$
Medlemstest	$\mathcal{O}(n)$	$\Theta(1)$
Ladda ordlista	$\Theta(n)$	$\Theta(n)$
Stavningskontroll	$\mathcal{O}(mn)$	$\Theta(m)$
$\approx$	$\mathcal{O}(n^2)$	$\Theta(n)$

Kan vi konstruera en sådan datastruktur?

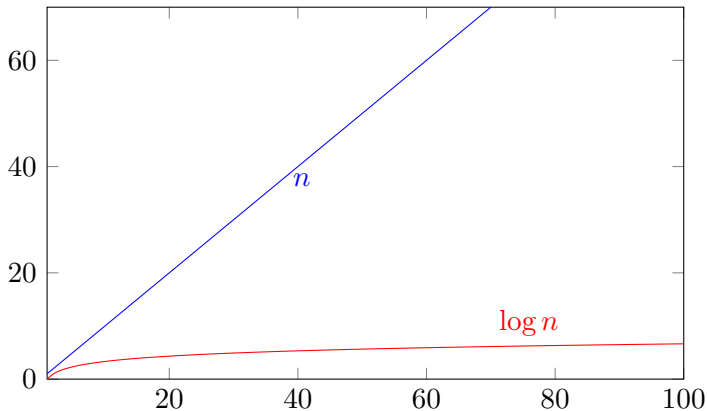
¹Medelfall

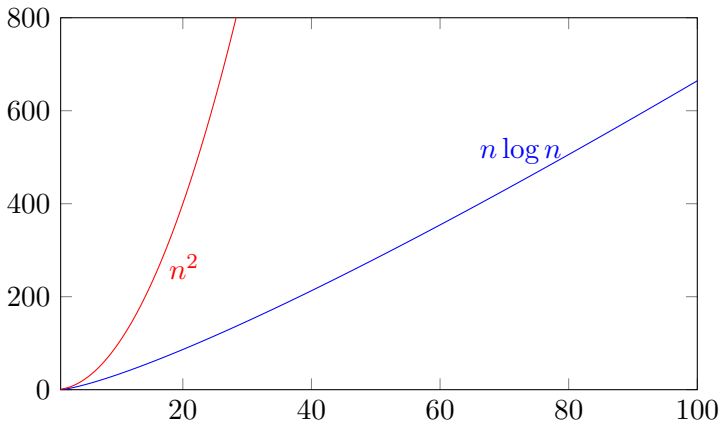
Analys – vad behövs?

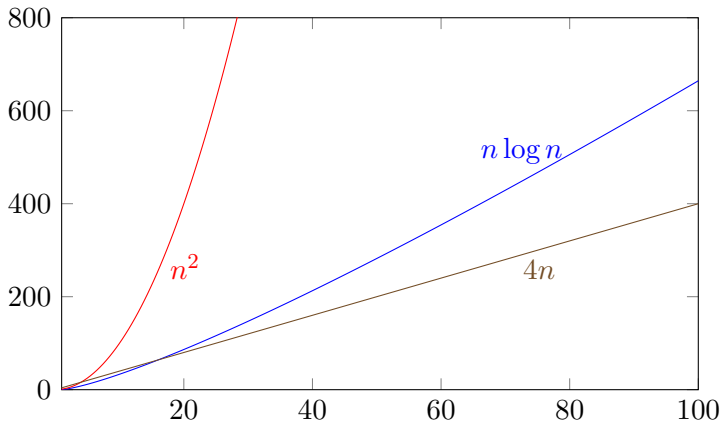
	vector	Önsketänkande	Mål idag
Insättning	$\Theta(1)^1$	$\Theta(1)$	$\mathcal{O}(\log(n))$
Medlemstest	$\mathcal{O}(n)$	$\Theta(1)$	$\mathcal{O}(\log(n))$
Ladda ordlista	$\Theta(n)$	$\Theta(n)$	$\mathcal{O}(n \log(n))$
Stavningskontroll	$\mathcal{O}(mn)$	$\Theta(m)$	$\mathcal{O}(m \log(n))$
$\approx$	$\mathcal{O}(n^2)$	$\Theta(n)$	$\mathcal{O}(n \log(n))$

Kan vi konstruera en sådan datastruktur?

¹Medelfall

$\Theta(n)$ vs. $\Theta(\log n)$ 

$\Theta(n^2)$ vs. $\Theta(n \log n)$ 

$\Theta(n^2)$ vs. $\Theta(n \log n)$ 

ADT dictionary (symboltabell)

En **oordnad** mängd av par, **nyckel-värde**.

Ofta ordnad efter **nyckel** på något sätt.

size() Antal element i datastrukturen

insert(k, v) Lägg till ett nyckel-värde-par

remove(k) Ta bort paret med nyckeln **k**

contains(k) Finns nyckeln **k**?

get(k) Hämta värdet för nyckeln **k**

ADT dictionary (symboltabell)

En **ordnad** mängd av par, **nyckel-värde**.

Ofta ordnad efter **nyckel** på något sätt.

size() Antal element i datastrukturen

insert(k, v) Lägg till ett nyckel-värde-par

remove(k) Ta bort paret med nyckeln **k**

contains(k) Finns nyckeln **k**?

get(k) Hämta värdet för nyckeln **k**

I C++: map, unordered_map, ...

Utan värde: set, unordered_set, ...

Vi testar set...

```
int main() {
    set<string> d = read_dict();

    int count{};
    string word;
    while (cin >> word) {
        if (d.find(word) == d.end())
            cout << "Wrong: " << word << endl;
        count++;
    }
    cout << "Checked " << count << " words" << endl;
    return 0;
}
```

Testkörning

```
$ time spellcheck < thesis.txt
99171 words in the dictionary
...
Checked 38040 words

real 0m0.475s
user 0m0.288s
sys 0m0.016s
```

Nu börjar det likna något!

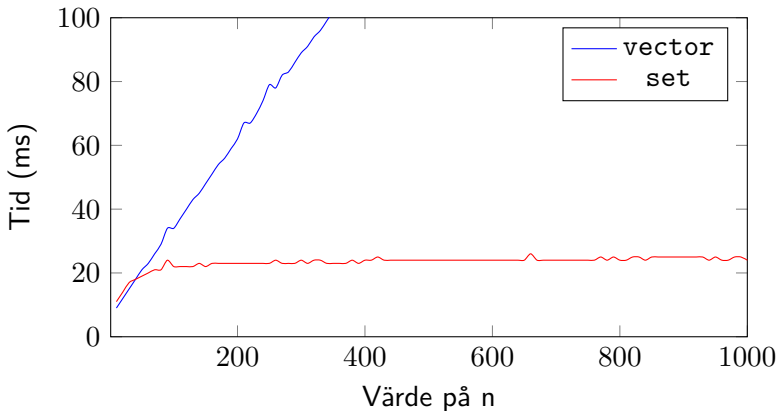
Analys

```
while (cin >> word) {  
    if (d.find(...) == d.end())  $\begin{matrix} O(?) \\ O(1) \end{matrix}$   
        cout << ...;  
}
```

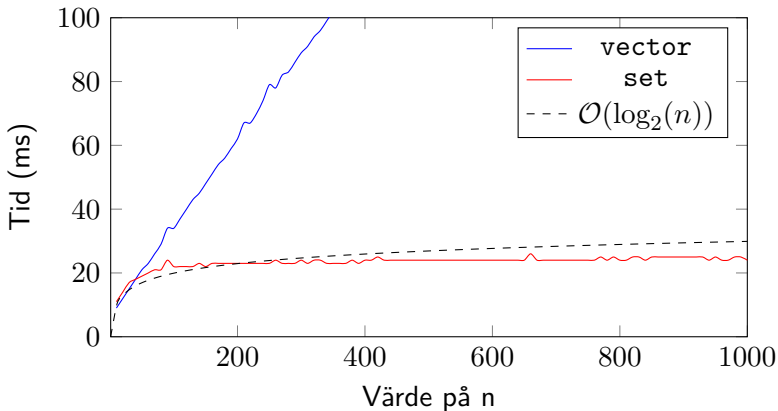
$\left. \vphantom{\begin{matrix} O(?) \\ O(1) \end{matrix}} \right) m \text{ gånger}$

Vad har find för komplexitet?

Vi testar! 100 000 körningar per indata



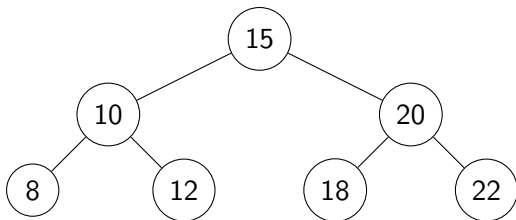
Vi testar! 100 000 körningar per indata



- 1 ADT dictionary (symboltabell)
- 2 **Binära sökträd**
- 3 AVL-träd, ett balanserat sökträd
- 4 Sammanfattning

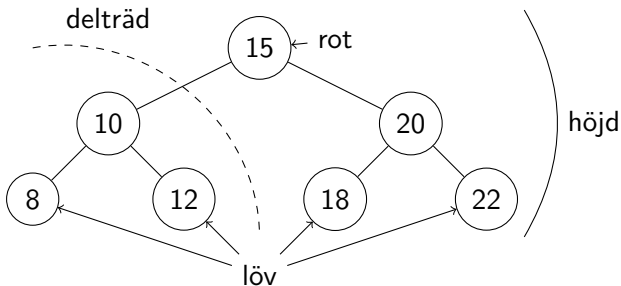
Hur är set implementerad?

Idé: Minimera söktid genom att använda ett sökträd!



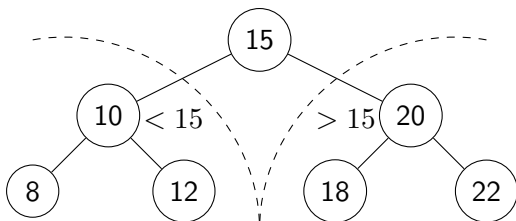
Hur är set implementerad?

Idé: Minimera söktid genom att använda ett sökträd!



Hur är set implementerad?

Idé: Minimera söktid genom att använda ett sökträd!



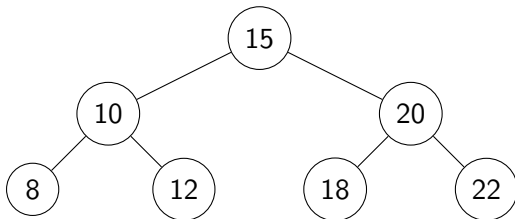
Hitta element i ett sökträd

```
class Node {  
public:  
    Node(int v, Node *l, Node *r) :  
        v{v}, l{l}, r{r} {}  
  
    int v;  
    Node *l;  
    Node *r;  
};
```

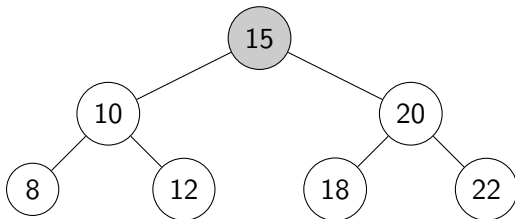
Hitta element i ett sökträd

```
Node *find(Node *root, int value) {  
    if (!root)  
        return nullptr;  
  
    if (value < root->v) {  
        return find(root->l, value);  
    } else if (value > root->v) {  
        return find(root->r, value);  
    } else { // value == root->v  
        return root;  
    }  
}
```

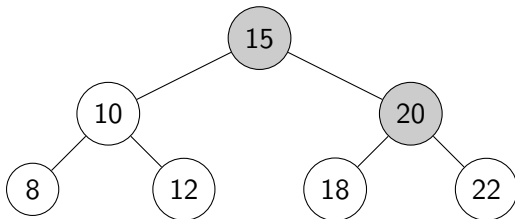
Exempel: Hitta 18



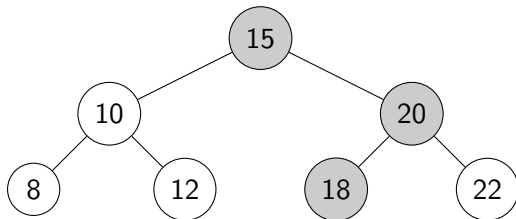
Exempel: Hitta 18



Exempel: Hitta 18



Exempel: Hitta 18



Komplexitet? Värsta och bästa fall?

Insättning

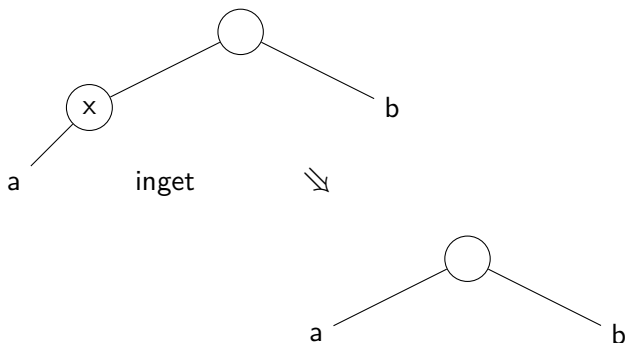
Insättning görs på liknande sätt. Testa exempelvis:

10, 5, 15, 20, 7, 12, 30, 17, 18, 40, 50, ...

Komplexitet?

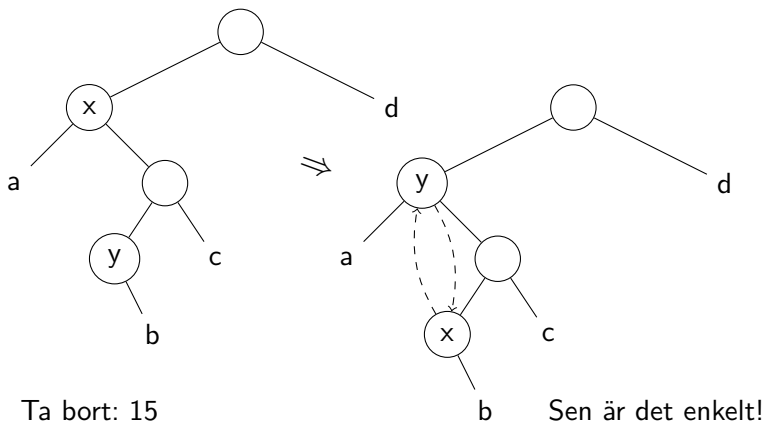
Vad kan vi säga om höjden av trädet?

Borttagning – enkelt fall



Ta bort: 7

Borttagning – generellt fall



Höjden av ett sökträd?

h : trädets höjd

n : antal noder i trädet

I värsta fall: $h = n$

I bästa fall: $n = 2^h - 1 \implies h = \log_2(n + 1)$

$\Rightarrow$ Vi vill se till att sökträdet är balanserat!

Tidskomplexitet hos ett BST

	Obalanserat	Balanserat
Insättning	$\mathcal{O}(n)$	$\mathcal{O}(\log_2(n))^2$
Medlemstest	$\mathcal{O}(n)$	$\mathcal{O}(\log_2(n))$
Ladda ordlista	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log_2(n))^1$
Stavningskontroll	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log_2(n))$

²Om det inte är dyrt att balansera trädet

- 1 ADT dictionary (symboltabell)
- 2 Binära sökträd
- 3 AVL-träd, ett balanserat sökträd
- 4 Sammanfattning

Balansering av träd

Sökträdet blir ineffektivt om det inte är balanserat.

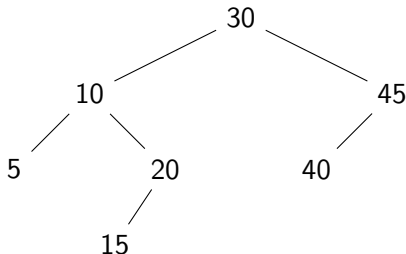
Idé: Håll koll på djupet i varje nod och "rotera" om det skiljer för mycket.

Viktiga frågor:

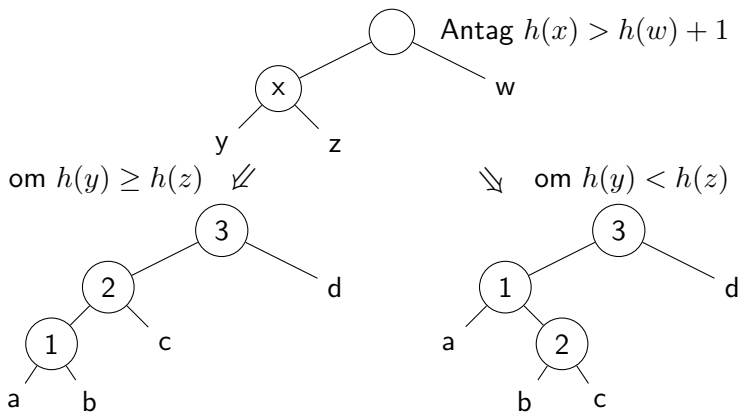
- Kan vi uppdatera höjderna inom tidsramarna?
- Vilka noder kan behöva "roteras"?
- Hur lång tid tar det?
- Hur stor obalans ska vi tolerera?

AVL-träd – Idé

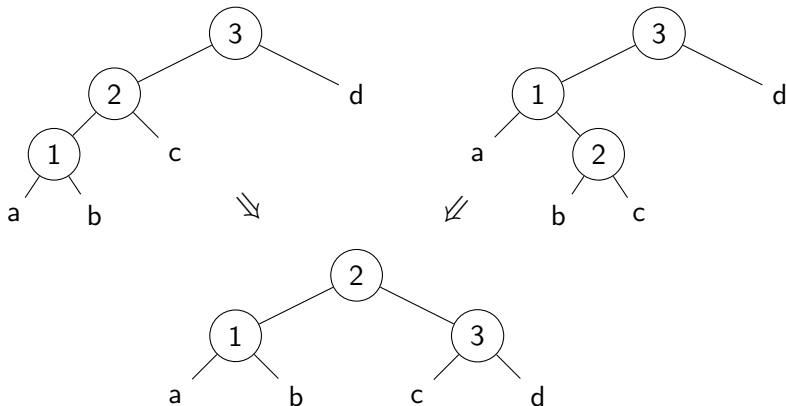
- Kom ihåg hur högt varje delträd är
- Skillnaden i höjd mellan syskon får maximalt vara 1
- *Rotera* noter vid insättning och borttagning



AVL-rotationer



AVL-rotationer



Insättning

Insättning görs som i sökträd, med balansering efteråt

Testa: 10, 5, 15, 20, 7, 30, 40, 35, ...

Tidskomplexitet?

Borttagning

Borttagning görs också som i sökträd, med balansering efteråt

Ta bort: 7, 10, 5, 15

- 1 ADT dictionary (symboltabell)
- 2 Binära sökträd
- 3 AVL-träd, ett balanserat sökträd
- 4 Sammanfattning

I kursen framöver

- Testlab
- Nästa föreläsning
 - ADT stack, kö, prioritetskö
 - Trädtraversering, heap, (fenwickträd)
- Extrauppgifter
 - 10295 (enkel)
Användning av en lämplig datatyp.
 - 10672 (svårare)
Övning på att traversera träd. Traversera inte onödigt många gånger!

Filip Strömbäck

www.liu.se